

# **Exhibit 5**

# **Exhibit 5**

# U P . L I N K D E V E L O P E R ' S G U I D E

V e r s i o n 1 . 0

---

UNWIRED PLANET, INCORPORATED  
390 Bridge Parkway  
Redwood Shores, California 94065  
USA

Part Number UDG-10  
July 1996



**IMPORTANT NOTICE**

---

Copyright 1996 Unwired Planet, Inc. All rights reserved.

These files are part of the Unwired Planet Software Developer's Kit (the "UP.SDK").

Subject to the terms and conditions of the UP.SDK License Agreement, Unwired Planet, Inc. hereby grants you authorization to use UP.SDK software and its related documentation.

TO THE MAXIMUM EXTENT PERMITTED BY LAW, UNWIRED PLANET MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED, REGARDING THE UP.SDK SOFTWARE, INCLUDING, BUT NOT LIMITED TO, ANY IMPLIED WARRANTIES OR MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NONINFRINGEMENT. USER UNDERSTANDS AND ACCEPTS THE UP.SDK SOFTWARE ON AN "AS IS" BASIS FROM UNWIRED PLANET, AND UNWIRED PLANET DOES NOT WARRANT, GUARANTEE, OR MAKE ANY REPRESENTATIONS REGARDING THE USE OF, OR THE RESULTS OF THE USE OF THIS UP.SDK SOFTWARE IN TERMS OF ITS CORRECTNESS, ACCURACY, RELIABILITY, OR OTHERWISE. IN NO EVENT SHALL UNWIRED PLANET BE LIABLE FOR ANY DAMAGES RESULTING FROM OR ARISING OUT OF YOUR USE OF THE UP.SDK SOFTWARE, INCLUDING, WITHOUT LIMITATION, ANY DIRECT, INDIRECT, CONSEQUENTIAL, OR PUNITIVE DAMAGES RESULTING FROM THE USE, MODIFICATION, OR DISTRIBUTION OF THE UP.SDK SOFTWARE OR ITS DERIVATIVES.

Unwired Planet, the Unwired Planet logo, UP.Phone, UP.Link, UP.Browser, UP.Access, UP.Mail, UP.Pager, HDML and UP.SDK are either trademarks or registered trademarks of Unwired Planet, Inc. in the United States.

## Contents

---

### **Preface v**

The UP.Link Developer's Kit v

UP.Link developer documentation vi

Style and typographical conventions vi

### **1 Introduction 1**

Overview of the UP.Link platform 1

Components of the UP.Link platform 1

Overview of UP.Link platform operation 7

Installing and setting up the UP.Link Developer's Kit 10

System requirements 10

UP.Link Developer's Kit files 10

Registering UP.Link services and phones 10

Example: creating and testing a simple UP.Link service 10

Creating a service 11

Running and testing the service 13

### **2 Using HDML 17**

HDML syntax 17

Structure 18

Options 18

Using decks with multiple cards 19

Handling user input 21

Using entry cards 21

Using choice cards 24

Building an argument list with multiple cards 25

Modifying deck navigation 29

Working with deck history and deck caching 29

Replacing a deck in the cache 33

Formatting display text 33

Adding line breaks 33

Setting text wrapping and horizontal scrolling 33

Specifying text alignment and tabs 34

Displaying special characters 35

Choosing the optimal deck size 36

<b>3</b>	<b>Creating UP.Link Services</b>	<b>39</b>
	Writing an application to implement a service	39
	Using HTTP headers	39
	Using URL arguments added by the UP.Link	40
	Generating compiled HDML	41
	Using Perl utilities to compile HDML	41
	Using the C++ HDML compiler library	42
	Using the command-line HDML compiler	44
	Retrieving web pages	45

## Preface

---

This manual provides instructions for developers who want to create services for the Unwired Planet UP.Link™ platform.

The UP.Link platform provides owners of data-capable handheld devices, such as cellular phones, with wireless access to a wide array of services. These services include, but are not limited to, many of the services available on the Internet.

An UP.Link-enabled handheld device uses the data capabilities of conventional cellular networks to connect to World Wide Web (WWW) servers through a system called an UP.Link. A software client, called an UP.Link browser™, in the device enables it to function much like a WWW browser. The user presses keys on the device to navigate and enter requests. The device sends the requests to an UP.Link, which converts them into Hypertext Transport Protocol (HTTP) requests.

The UP.Link sends HTTP requests over the Internet or direct lines to web servers maintained by Unwired Planet or third-party developers. The applications on these web servers that handle UP.Link requests are called UP.Link services. UP.Link services respond to the requests using Handheld Device Markup Language (HDML), an open language developed by Unwired Planet to interactively display information on handheld devices. When an UP.Link receives a response from an UP.Link service, it relays it back to the device that originated the request. The device displays the response to the user.

## The UP.Link Developer's Kit

---

The UP.Link Developer's Kit provides tools for developers who want to create or maintain UP.Link services. It includes the following components:

- The UP.Phone simulator for Windows 95 and Windows NT, which simulates the behavior of an UP.Link-enabled device and makes it easy to test UP.Link services
- Libraries of Perl functions, which simplify the process of generating HDML and handling HTTP requests
- The HDML compiler, which compiles HDML into a compact format that UP.Link-enabled devices can interpret

- UP.Link developer documentation, which consists of online manuals provided in Adobe Acrobat Portable Document Format (PDF)

## UP.Link developer documentation

---

UP.Link developer documentation assumes that you are familiar with the Common Gateway Interface (CGI) protocol and the Perl programming language. To get the most from UP.Link developer documentation, you should also be familiar with the HyperText Markup Language (HTML) and the C++ programming language.

UP.Link developer documentation includes the following manuals:

- The *UP.Link Developer's Guide*, the manual you are reading now, provides instructions for getting started with the UP.Link Developer's Kit. It discusses the basics of generating HDML and providing an UP.Link service. It is recommended that you read this manual first.
- The *HDML Language Reference* provides reference information about the HDML language and the UP.Link Developer's Kit. It assumes you have already read the *UP.Link Developer's Guide* and are familiar with the UP.Link Developer's Kit.

UP.Link developer documentation is provided in PDF format so that you can view and print it easily. For more information on using PDF documents, see the Adobe Systems Corporation web site at: <http://www.adobe.com>.

## Style and typographical conventions

---

UP.Link developer documentation distinguishes between *you*, the developer, and *the user*, the person who uses the services you create.

The term *UP.Phone* or *phone* refers to all UP.Link-enabled devices, such as cellular phones, Personal Digital Assistants (PDAs), and two-way pagers. It includes both the device hardware and the UP.Link Browser client software from Unwired Planet that is installed on it.

UP.Link developer documentation and the HDML language use the names ACCEPT, CANCEL, and SOFT1 to specify function keys on the UP.Phone. The actual labels and locations of these keys vary from one device to another. For more information on handheld device configurations, see "UP.Link-enabled devices" in Chapter 1 of the *UP.Link Developer's Guide* and the Unwired Planet developer's web site at: <http://www.uplanet.com>.

The phone display examples in this manual depict a 4 X 20 character display on a device that has three function keys. Output will appear differently on a device that has a different display size or a different number of function keys. HDML is designed to work with phone displays as small as 3 X 12.

Omitted code in examples is indicated with ellipses. For example, the ellipses in the following HDML code example indicate that some of the code necessary for a complete HDML deck has been omitted:

```
<HDML VERSION=0.1.0>  
.  
.  
.  
</HDML>
```

Some code listings in this manual have line numbers, which appear to the left of the code. For example:

```
1 <HDML VERSION=0.1.0 DECKNAV=NOAUTOLOAD>  
2   <DISPLAY>  
3     A deck.  
4   </DISPLAY>  
5 </HDML>
```

These line numbers are used for reference purposes only and are not part of the actual code.

This manual uses different fonts to represent different types of information.

- Text that you enter, function names, filepaths, and Uniform Resource Locators (URLs) appear in `text like this`.
- Required HDML keywords and characters in syntax descriptions appear in `text like this`.
- *Optional* HDML keywords and characters in syntax descriptions appear in *text like this*.
- Placeholders that you replace with your own text appear in *text like this*.





# 1

## Introduction

---

This chapter provides an overview of the UP.Link platform. It briefly describes how to install and set up the UP.Link Developer's Kit and provides instructions on how to get an UP.Link service running right away.

## Overview of the UP.Link platform

---

This section describes the UP.Link platform components and provides an overview of how they operate together.

### Components of the UP.Link platform

---

The UP.Link platform includes the following components:

- UP.Phones—handheld devices that replace conventional web browsers
- The HDML language—a set of commands and statements that specify how an UP.Phone interacts with a user
- UP.Links—which relay messages between UP.Phones and web servers on conventional networks, such as the Internet
- UP.Link services—which are web server applications that handle the requests relayed by UP.Links

Figure 1-1 summarizes the interactions between these components.

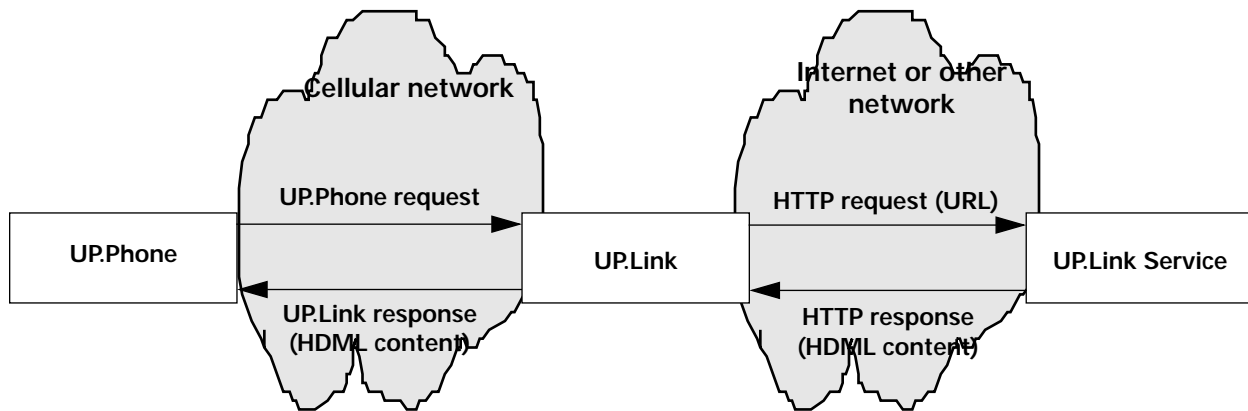


FIGURE 1-1. UP.Link platform components

The following sections describe each component of the UP.Link platform in greater detail.

### UP.Phones

An UP.Phone is a data-capable handheld device running the UP.Link Browser. The UP.Link Browser is a software module that enables the handheld device to interpret HDML and communicate with the UP.Link.

A variety of handheld devices can be UP.Link enabled. For a list of supported models and capabilities, see the Unwired Planet web pages located at:

<http://www.uplanet.com>

Because the UP.Link platform supports a variety of device models, this manual uses the concept of a generic, or abstract, UP.Phone which has the minimal set of capabilities that most handheld devices share. HDML syntax descriptions and examples in this manual refer to this abstract phone. The abstract phone has the following characteristics:

- A 4 X 12 fixed-width character display that can be scrolled vertically
- Support for numeric data entry
- Support for alphabetic data entry
- Backspace editing
- An accept key, which this manual refers to as ACCEPT
- A cancel key, which this manual refers to as CANCEL
- One softkey, which this manual refers to as SOFT1

Some features of the abstract phone may not be implemented as hardware on all handheld device models. For example, on some models, SOFT1 is an actual physical key; on other models, it is just a text line appended at the end of display.

An UP.Phone behaves much like a conventional web browser. Its interface (the text it displays and the input options it presents to the user) is controlled by the sets of commands it loads. For a WWW browser, the command language is HTML; for an UP.Phone, it is HDML—a language optimized for the constraints of small handheld device displays and limited keypads. For a WWW browser, the sets of commands are called web pages, for an UP.Phone they are called *decks*.

The user navigates with the UP.Phone by pressing keys on the UP.Phone keypad. As the user navigates to different locations, the phone loads different decks, which change the display and the input options. Like a WWW browser, an UP.Phone maintains a cache of locations the user has visited. Unlike a WWW browser, the UP.Phone does not allow the user to set the cache size. Instead, the cache is constrained by the handheld device's available memory and by HDML command options that specify how long specific decks should be cached.

### The HDML language

The HDML language provides a set of commands or statements that specify how an UP.Phone interacts with a user. HDML statements display information on a phone, provide input options for the user, and specify how the phone responds when the user presses keys. For example, an HDML statement can instruct a phone to display a prompt and allow the user to enter text.

The deck is the fundamental unit of HDML. Each group of instructions the UP.Link relays from an UP.Link service to a phone is in the form of a single deck. A deck contains one or more *cards*, each of which specifies a single interaction between the phone and the user. HDML currently supports the following types of cards:

- *Display cards*—which simply display information
- *Entry cards*—which display a message and allow the user to enter a string of text
- *Choice cards*—which display a list of options from which the user can choose a single option

The following HDML statements are an example of a simple HDML deck containing a single display card:<sup>1</sup>

```
1 <HDML VERSION=0.1.0 DECKNAV=NOAUTOLOAD>
2   <ACTION TYPE=ACCEPT GO=:1000?page=2>
3   <DISPLAY>
4     Hello Unwired World!
5   </DISPLAY>
6 </HDML>
```

Line 1 is the deck header. `VERSION` and `DECKNAV` are deck options, which will be discussed later in this manual.

Line 2 specifies what the phone does when the user presses the `ACCEPT` key while the deck is loaded in the phone. In this example, the phone requests service number 1000 with the option, `page=2`.<sup>2</sup> Service numbers are described in “UP.Link services” on page 5.

Lines 3-5 define the display card. They simply instruct the phone to display the text: Hello Unwired World!

Line 6 is the deck footer. The phone treats everything between the deck header and the deck footer as a single HDML deck.

When a phone loads this deck, its display appears as shown in Figure 1-2.

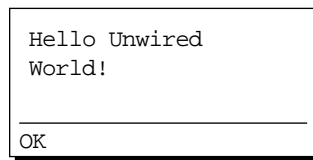


FIGURE 1-2. An HDML deck with a display card

## The UP.Link

The UP.Link is the backbone of the UP.Link platform. The UP.Link maintains a database of phone subscribers and UP.Link services, which it uses to route phone requests to services and service responses back to phones.

For a phone to work with an UP.Link, it must be registered in the UP.Link's database. For a service to receive phone requests from an UP.Link, its URL must be registered in the UP.Link database. For information on registering services and phones, see “Registering UP.Link services and phones” on page 10.

1. Line numbers in this and other HDML examples in this manual are for reference only and are not part of the actual HDML code.
2. For the sake of simplicity, this example specifies the actual service number. Normally, you should use just a question mark (?) instead of the actual service number to request the current service. For example, you could use: `GO=?page=2`. The UP.Phone automatically converts requests to use actual service numbers. So, if the current service number is 1000, `GO=:1000?page=2` and `GO=?page=2` have the same effect. By not specifying an actual service number in your HDML, you avoid problems that may arise if you need to change the service number.

The UP.Phone uses an extended URL format that differs slightly from the conventional URL format. This format is part of a protocol called *UP.Link Gateway Protocol (UGP)*. For example, when the user presses the ACCEPT key in the simple deck described in the previous section, the UP.Phone sends the following request to the UP.Link:<sup>3</sup>

```
:1000?page=2
```

The first part of the phone request (:1000) specifies a service number. This is the number the UP.Link assigns to a service when you register it. The second part of the request (?page=2) specifies an option. The UP.Link converts the phone request into a conventional URL by mapping the service number to the service's URL and appending the specified option (or options) to it. For example, if the URL for service 1000 is:

```
http://jkh.com/cgi-bin/test.cgi
```

the UP.Link converts the phone request :1000?page=2 into the following URL:

```
http://jkh.com/cgi-bin/test.cgi?page=2
```

The following steps summarize how the UP.Link handles a phone request:

- 1 It checks its database to make sure the phone is registered. If the phone is not registered, it returns an error message to the phone.
- 2 It converts the phone request to a URL.
- 3 It issues an HTTP request for the URL.
- 4 It waits for an HTTP response. If the response does not contain valid HDML, it returns an error message to the phone.
- 5 It sends the HDML to the phone.
- 6 It logs the transaction.

### UP.Link services

UP.Link services are web server applications that return HDML content in response to HTTP requests. It is possible for an UP.Link service to provide only a static HDML deck. However, the small size and portability of UP.Phones makes them most useful for viewing timely, dynamic information. So UP.Link services are normally CGI applications that generate HDML decks dynamically. Some examples of UP.Link services are:

- Securities trading systems
- Order tracking systems
- Weather and road condition reports

---

3. Actually, the phone includes this request in a wrapper. However, the wrapper format is internal to the UP.Link platform and it is not necessary for UP.Link developers to understand.

An UP.Link service can dynamically generate HDML in the same way conventional CGI scripts generate HTML—for example, by querying databases or other external data sources. It is also easy for an UP.Link service to generate HDML from existing HTML web pages. The UP.Link Developer's Kit provides PERL utilities that help you retrieve HTML web pages.

You can register your service with Unwired Planet for free. However, if you want to limit access to your service or charge users for it, you must make arrangements with Unwired Planet and pay a fee.

To create an HTTP response to an UP.Link request, an UP.Link service does the following:

- 1 Generates a complete HDML deck.

An UP.Link request looks like an ordinary HTTP request, so the web server handles it in the same way: it calls the service to process it. If an UP.Link service is a CGI application, the web server passes arguments to it by setting environment variables. The service generates HDML by checking the environment variables and querying external data sources.

- 2 Uses the HDML compiler to compile the HDML deck.

The HDML compiler condenses the HDML into a more compact form that is more efficient for transmission over the Internet and wireless networks. The UP.Link Developer's Kit includes a command-line version of the HDML compiler (`hdmlc`), which you can use in Perl scripts, and a C++ class library version that you can use in C++ applications.

- 3 Prepends an HTTP header to the beginning of the compiled deck.

The HTTP header specifies the content type of the response. It should appear as follows:

```
HTTP/1.0 200
Content-type: application/x-hdmlc
```

---

**IMPORTANT** If the service uses Netscape CGI conventions, it should omit the status portion of the header (`HTTP/1.0 200`).

---

For example, a service could generate the following HDML deck in response to an HTTP request from an UP.Link:

```
<HDM1 VERSION=0.1.0>
  <DISPLAY>
    A very simple HDM1 deck.
  </DISPLAY>
</HDM1>
```

When the service runs `hdmlc` on this deck, it generates compiled binary HDML, which appears similar to the following text when it is printed:

ÃA©very©simple©HDMLEdeck.

The complete HTTP response that the service sends to the UP.Link appears similar to the following:

Content-type: application/x-hdmlc

EãA©very©simple©HDMLEdeck.

## Overview of UP.Link platform operation

The architecture of the UP.Link platform leverages the design and infrastructure of the WWW. The basic operation of the WWW is as follows. Suppose a user using a conventional web browser chooses the URL, `http://jkh.com/dir/jkh.html`. **Figure 1-3** summarizes the resulting interaction.



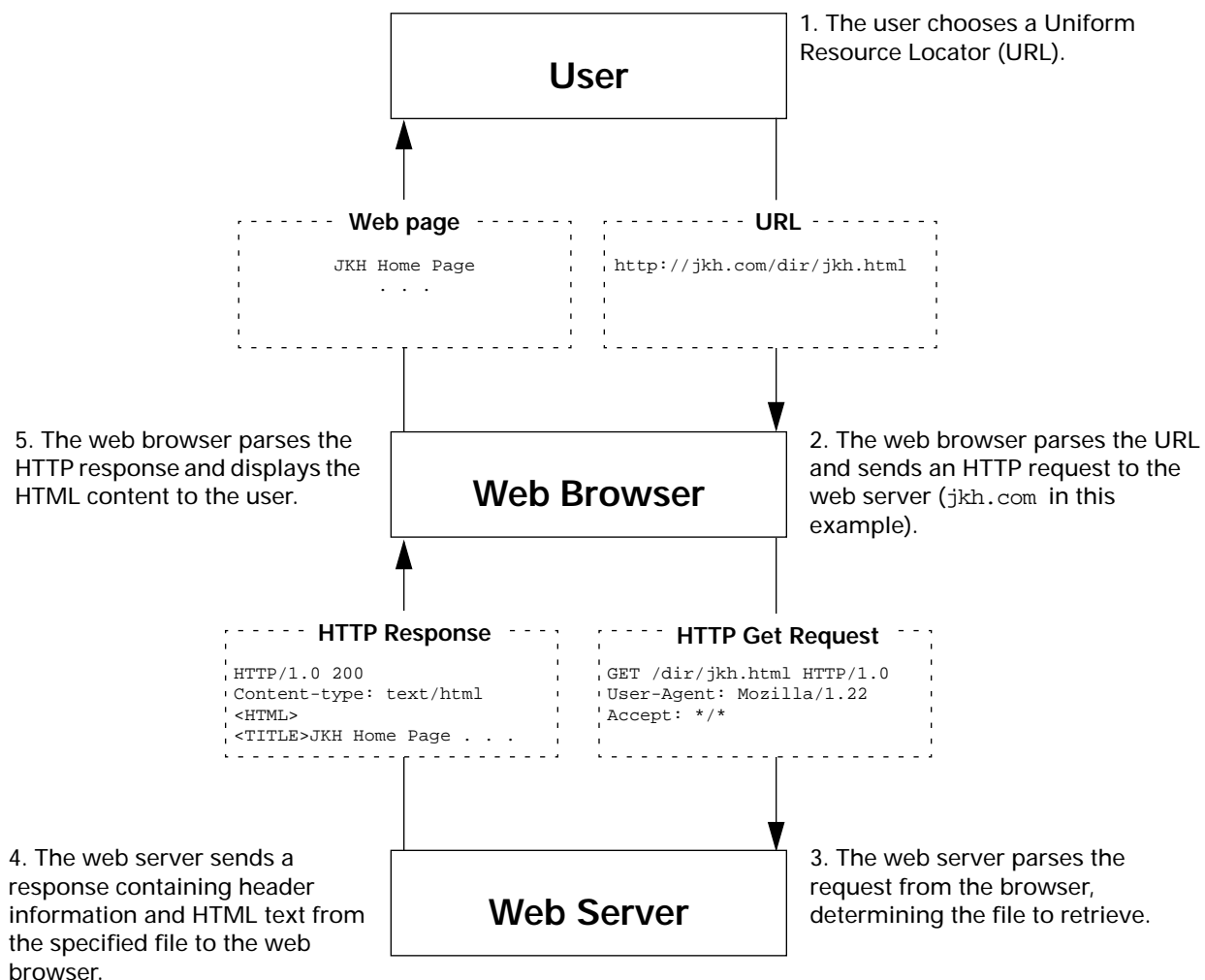


FIGURE 1-3. Steps in a request for a WWW page

The UP.Link platform uses the WWW model, but replaces the web browser with the combination of the UP.Phone and the UP.Link. Suppose a user presses ACCEPT when the phone is displaying the following deck:

```
<HDML VERSION=0.1.0 DECKNAV=NOAUTOLOAD>
  <ACTION TYPE=ACCEPT GO=:1000?page=2>
  <DISPLAY>
    Hello Unwired World!
  </DISPLAY>
</HDML>
```

Assuming the URL registered for service 1000 is `http://jkh.com/jkh.cgi`, Figure 1-4 summarizes the resulting interaction.

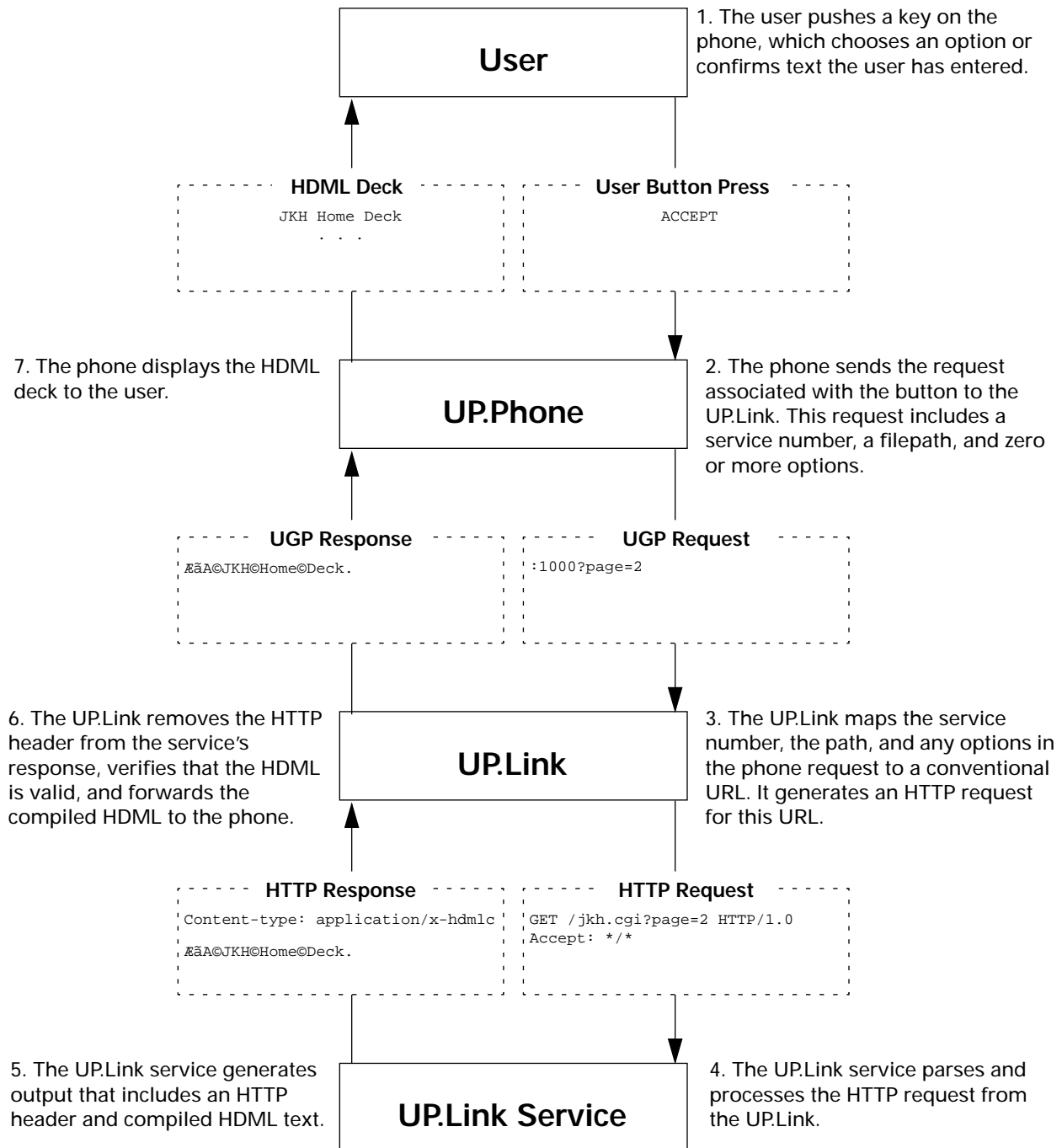


FIGURE 1-4. Steps in a request from an UP.Phone

---

## Installing and setting up the UP.Link Developer's Kit

---

This section describes system requirements for the UP.Link Developer's Kit. It also describes where to get the UP.Link Developer's Kit components you need and how to set them up.

### System requirements

---

The UP.Link Developer's Kit has the following system requirements:

- To provide an UP.Link service, you need an HTTP-compliant web server linked to the Internet or directly connected to an UP.Link
- To run the phone simulator application, your computer must be running the Windows NT or Windows 95 operating system
- To use the phone simulator to test a service through an UP.Link, your computer must have an Internet connection

### UP.Link Developer's Kit files

---

To get the most recent copies of UP.Link Developer's Kit files, check the Unwired Planet web site at:

<http://www.uplanet.com>

### Registering UP.Link services and phones

---

To test your service with the phone simulator, you must register the service and the phone simulator on an UP.Link. For information on registering the service and the phone simulator on the Unwired Planet developer UP.Link, see the `README.TXT` file included with the UP.Link Developer's Kit. To register the service and the phone simulator on a private UP.Link, contact the UP.Link administrator.

---

## Example: creating and testing a simple UP.Link service

---

This section is for developers who want to get a simple UP.Link service up and running right away. If you feel you need some more background on the UP.Link platform before you get started, skip this section for now and come back to it after reading chapters 2 and 3 of this manual.

## Creating a service

Suppose you want to provide a wave condition service for surfers. Florida State University provides web pages with continuously updated, real-time wave information from National Oceanographic and Atmospheric Administration (NOAA) marine buoys. For example, the web page for the Santa Cruz, California buoy contains HTML code similar to the following:

```
<html>
<head>
<META HTTP-EQUIV="Refresh" Content=360>
<title>Current Observation for Station 46042</title>
. . .
<h2>Oceanographic Data</h2>
Sea Surface Temperature:      53.1 F<br>
Wave Height:      8.5 ft.<br>
. . .
```

The following Perl code retrieves this web page, reformats its data into HDML, and generates an HTTP response. Following the code is a line-by-line explanation of how it works.

```
1  #!/usr/local/bin/perl
2  #
3  require 'HDMLOutput.pl';
4  require "HTTPClient.pl";
5
6  &AppUtils'HTTPConnect(*HTTP,
7      "http://www.met.fsu.edu/nws/cgi-bin/buoy.cgi?46042");
8  while(<HTTP>)
9  {
10     if(/Wave Height/i)
11     {
12         s/<br>//i;
13         $height = $_;
14         last;
15     }
16 }
17 if(!defined($height))
18 {
19     $height = "Wave height not available";
20 }
21 &AppUtils'HTTPDisconnect(*HTTP);
22 $DECK = "<HDML VERSION=0.1.0 TTL=0>".
23     "<DISPLAY>".
24     "<CENTER>Santa Cruz".
25     "<WRAP>$height".
26     "</DISPLAY>".
27     "</HDML>";
28
29 &AppUtils'OutputDeck($DECK);
```

**Line 1**

This line identifies the Perl version to use. Change this line as needed for your server.

**Lines 3-4**

These lines include the `HTTPClient.pl` and `HDMLOutput.pl` utility files provided in the UP.Link Developer's Kit. These files define the `HTTPConnect()`, `HTTPDisconnect()`, and `OutputDeck()` functions, which the application uses to retrieve web pages and output HDML.

**Lines 6-7**

These lines use the `HTTPConnect()` utility function to connect a socket to the web page containing the wave data. The function reads the HTTP status line but leaves the rest of the output.

**Lines 8-20**

These lines search the HTML of the retrieved web page for the line containing the wave height information (the line starting with the string, `Wave Height`).

The line in the web page ends with a `<br>` statement. This statement has the same effect in HDML as it does in HTML: it instructs the phone to add a line break. Because an extra line break at the end of the display card is not needed, Line 12 removes the statement. Normally, you should remove all HTML statements and tags from text that you put in an HDML deck. If the HDML deck contains a statement or tag that is not valid HDML, it fails and generates an error message. For a list of HDML compiler error messages, see the *HDML Language Reference*.

If the line containing the wave height information isn't found, an error message is stored to the variable that gets displayed on the phone.

**Line 21**

This line uses the `HTTPDisconnect()` utility function to disconnect the socket from the web page.

**Line 22**

This line defines the header of the HDML deck to be generated. The `VERSION` option in the `<HDML>` statement specifies the HDML version the deck uses (0.1.0). The `TTL` option specifies how long, in seconds, the phone should cache the deck. Because the wave information is extremely timely, the option specifies 0—which means the phone doesn't cache the deck at all.

### Lines 23-26

These lines define a display card that displays the string `Santa Cruz` and the line from the web page that provides the wave height. The `<CENTER>` tag instructs the phone to center the string `Santa Cruz`. The `<WRAP>` tag instructs the phone to wrap the string that provides the wave height onto the next line if it exceeds the display width.

### Line 29

This line uses the `OutputDeck()` utility function to send the deck as an HTTP response. The `OutputDeck()` function compiles the HDML and adds the HTTP header to it. It has the same result as the following lines of code:

```
print "Content-type: application/x-hdmlc\r\n";
open (HDMLOUT, "./|hdmlc");
select HDMLOUT;
print $DECK;
```

## Running and testing the service

The following sections describe how to run and test the service.

### Testing the service by itself

Before you register a service like the one in this example, test the service by itself. It is usually easier to isolate errors in your code by testing it by itself before testing it through the UP.Link.

To test the service by itself, follow these steps:

- 1 **Make sure the files that the service's script references are available to the script.** Make sure the script correctly references the Perl utilities files `HTTPClient.pl` and `HDMLOutput.pl`. Either put the files in the same directory as the script or edit the script to specify the full filepath of the files. If you are using UNIX, make sure the directory containing `hdmlc` is in your path. If you are using Windows NT, make sure the directory containing `hdmlc.exe` is in your path.
- 2 **Make sure the script's file permissions are set so that it is executable.**
- 3 **Make sure the computer you are running the script on is connected to the Internet.**
- 4 **Set the `QUERY_STRING` environment variable to set the arguments the script uses.**

The example provided above does not process any CGI arguments. However, if your script does process CGI arguments, you may want to test it with different argument values. To do this, set the `QUERY_STRING` environment variable with arguments that would normally be appended to your service's URL in an HTTP request. If you are using functions in the `CGISupport.pl` utilities file, you should also set the `REQUEST_METHOD` environment variable to `GET`.

- 5 **Run the script using your operating system's command-line interface.**  
For example, on UNIX, type the script's name in a command shell.

The script should generate an HTTP header followed by some binary code. It should look something like the following:

```
Content-type: application/x-hdmlc
```

```
Ñ 0±Santa@CruzWave@Height:©8.5©ft
```

Because compiled binary HDML is not very easy to read, you may want to temporarily modify the script to generate uncompiled HDML. To do this, change line 29 from:

```
&AppUtils'OutputDeck($DECK);
```

to:

```
&AppUtils'OutputDeck($DECK, 1);
```

Adding a nonzero second argument to the call instructs `OutputHDML()` to generate uncompiled HDML. After you make this change, the output from the script should look similar to the following:

```
Content-type: application/x-hdmlc
```

```
<HDML ttl=0 version=0.1.0>
  <DISPLAY>
    <CENTER>Santa Cruz
      <WRAP>Wave Height: 8.5 ft.
    </DISPLAY>
  </HDML>
```

If the output does not appear like this, you may not have access to the wave report web page. Use a conventional web browser, such as Netscape and make sure you can load the following URL:

```
http://www.met.fsu.edu/nws/cgi-bin/buoy.cgi?46042
```

### Testing the service with the phone simulator

The phone simulator allows you to test your services through the UP.Link. With the phone simulator, you can see exactly what an UP.Link subscriber sees on an UP.Link-enabled handheld device.

To test your service with the phone simulator, follow these steps.

- 1 **Register your service with an UP.Link.**  
For information on registering a service, see "Registering UP.Link services and phones" on page 10. When you register your service, you assign it a service number. You'll need this number later, so be sure to write it down.

**2 Register your phone simulator with the UP.Link.**

If your phone simulator is not already registered, you must register it to test your service through the UP.Link. For information on registering a service, see “Registering UP.Link services and phones” on page 10.

**3 Start the phone simulator application provided with the UP.Link Developer’s Kit.**

The UP.Link Developer’s Kit includes only a Windows NT version of the phone simulator. See the `README.TXT` file included in the UP.Link Developer’s Kit for the phone simulator’s executable filename and description.

If the phone simulator correctly connects to the UP.Link, after a few moments, it displays a menu similar to the one shown in Figure 1-5.

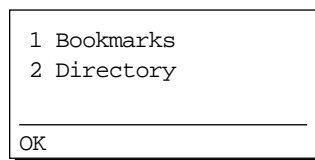


FIGURE 1-5. Initial phone simulator display

When you start the phone simulator, it opens a shell window, in which it displays status messages. Ignore this window for now.

**4 Navigate to your service.**

Click 2 on the phone simulator to choose the Directory option, then press the ACCEPT key. Another list of options appears. Scroll down to the Find option and press ACCEPT again. Scroll down to the By Number option and press ACCEPT again. The phone prompts you for a service number. Enter the service number that you jotted down in Step 1. After a few moments, the phone should display the message from your service as shown in Figure 1-6.

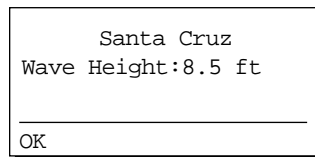


FIGURE 1-6. Display with a simple message from an UP.Link service

The functionality you eventually want your service to provide is undoubtedly more complex than what this very simple example provides. However, you will still follow the basic steps described in this section to test your service.





## Using HDML

---

# 2

This chapter provides instructions for using the HDML language. It includes an overview of HDML syntax and demonstrates how to use HDML statements to build your UP.Link service's interface.

### HDML syntax

---

This section provides a brief overview of HDML syntax. For a more complete description of HDML syntax and detailed reference information on the HDML language, see the *HDML Language Reference* provided with the UP.Link Developer's Kit.

HDML is a relatively simple Standard Generalized Markup Language (SGML) variant similar to HTML. It consists of angle bracket-delimited statements that define interactions between a user and an UP.Phone.

Like HTML, HDML uses the ASCII character set. The HDML compiler ignores the case of HDML keywords. It also converts one or more contiguous newlines, carriage returns, tabs, or spaces to a single space. The HDML examples in this manual are formatted with newlines and tabs to make them easier to read. However, this formatting is not required for the HDML to be valid. In fact, it is removed by the HDML compiler.

## Structure

Each HTTP message your service sends to the UP.Link must contain exactly one `<HDML>` statement. The high-level syntax for the `<HDML>` statement is:

```
<HDML VERSION=version_num deck_options>
    actions
    cards
</HDML>
```

Element	Meaning
<b>VERSION=version_num</b>	Specifies the HDML version number, such as "0.1.0". This option is required.
<i>deck_options</i>	Specify options that modify how the phone handles the deck. These options are discussed in more detail in "Working with deck history and deck caching" on page 29 and the <i>HDML Language Reference</i> .
<i>actions</i>	Specify actions associated with phone function keys while the deck is loaded in the phone. These actions include tasks such as requesting another deck.
<i>cards</i>	Include one or more <code>&lt;DISPLAY&gt;</code> , <code>&lt;CHOICE&gt;</code> , or <code>&lt;ENTRY&gt;</code> statements that define individual cards. The syntax for these statements is provided later in this chapter and in the <i>HDML Language Reference</i> .

Note that this manual uses the following style conventions in syntax statements: HDML keywords appear in bold; placeholders that you replace with your own text appear in italic; optional keywords appear in italic bold.

## Options

Many HDML statements allow you to specify options. Options appear before the closing angle bracket of an HDML statement and have the following syntax:

```
option1=value1 option2=value2 option3=value3 . . .
```

Each option-value pair is separated by white space. If the value contains nonalphanumeric characters, you must enclose it in double quotation marks ("). Whitespace (tab, newline, carriage return, and space) characters are not allowed between the option, the equal sign, and the value. For a complete list of the available options for an HDML statement, see the *HDML Language Reference*.

---

## Using decks with multiple cards

---

The sample decks in Chapter 2 of this manual contain only a single display card. It is possible for a single display card to contain a large amount of information. When a card has more lines than the phone can display at once, the user can scroll to see all of the lines. However, when you want to display more than a relatively small amount of information, it is recommended that you divide it into multiple display cards, rather than putting it all on one card. This enables the user to navigate quickly through the information.

Before defining a deck containing multiple display cards, it is helpful to look at the complete syntax of the `<DISPLAY>` statement:

```
<DISPLAY name=card_name>
  actions
  display_text
</DISPLAY>
```

Element	Meaning
<i>name=card_name</i>	Specifies a name that allows you to navigate to the card from other cards in the same deck.
<i>actions</i>	Specify actions to execute when the user presses a function key. Actions include tasks such as requesting another deck or displaying another card in the current deck.
<i>display_text</i>	Specifies the text the card displays.

The following deck displays a weather forecast on three separate display cards. The display cards are shown in Figure 2-1.

```
<HDML VERSION=0.1.0 DECKNAV=NOAUTOLOAD>
  <DISPLAY NAME=currfcst>
    <ACTION TYPE=ACCEPT LABEL=Tues GO="#tuesfcst">
      Current temps
      <BR>Hi: 60
      <BR>Lo: 28
    </DISPLAY>
  <DISPLAY NAME=tuesfcst>
    <ACTION TYPE=ACCEPT LABEL=Wed GO="#wedfcst">
      Tuesday temps
      <BR>Hi: 78
      <BR>Lo: 36
    </DISPLAY>
  <DISPLAY NAME=wedfcst>
    <ACTION TYPE=ACCEPT LABEL=Today GO="#currfcst">
      Wednesday temps
      <BR>Hi: 68
      <BR>Lo: 26
    </DISPLAY>
</HDML>
```

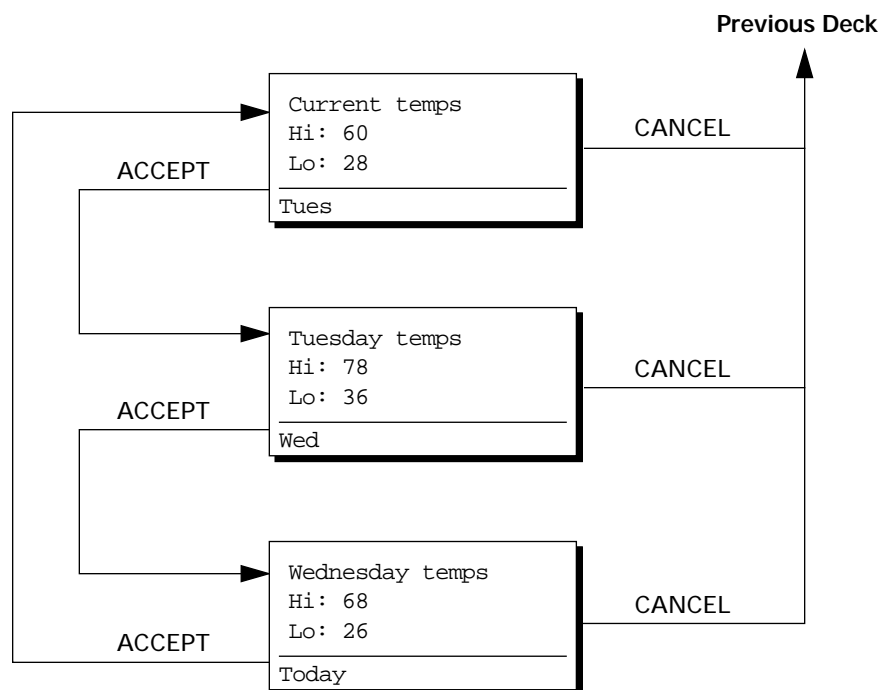


FIGURE 2-1. Multiple display cards in a deck

When a phone first loads a deck, it automatically displays the first card in the deck. The order in which it displays subsequent cards depends on what the user does and on the navigation scheme provided by the deck.

The deck defined above provides the simple navigation scheme indicated by the arrows in Figure 2-1. When the user presses ACCEPT in either of the first two cards, the phone displays the next card. When the user presses ACCEPT in the last card, the phone displays the first card again. This navigation scheme is specified by the `<ACTION>` statements in the display cards. For example, the first card's action statement:

```
<ACTION TYPE=ACCEPT LABEL=Tues GO="#tuesfcst">
```

instructs the phone to change the label of the ACCEPT key from *OK* to *Tues* and to display a card named `tuesfcst` (the second card in the deck) when the user presses the key.

You can define `<ACTION>` statements at several levels in HDML. Chapter 1 provided several examples in which `<ACTION>` statements were defined at the deck level. The previous example defines them at the card level. An `<ACTION>` statement's scope covers the HDML element in which it is defined: if it is defined in a deck's `<HDML>` statement, it applies to the whole deck; if it is defined in a card's `<DISPLAY>`, `<ENTRY>`, or `<CHOICE>` statement, it applies only to the individual card.

When one `<ACTION>` statement falls within the scope of another, the statement with the narrower scope takes precedence. For example, if we add an ACCEPT `<ACTION>` statement at the deck level in the example above, it has no effect, since each card in the deck also defines an ACCEPT `<ACTION>` statement and those statements take precedence.

Each of the phone function keys also has a default action associated with it. For the ACCEPT and CANCEL keys, the default action is to return to the previous deck; for the SOFT1 key, it is no response. For example, in the deck defined above, there are no `<ACTION>` statements specifying behavior for the CANCEL key. So when the user presses CANCEL anywhere in the deck, the phone displays the previous deck.

## Handling user input

HDML supports two kinds of interactive cards: entry cards and choice cards. The following sections describe how to use these cards to prompt the user for input and how to handle the input.

### Using entry cards

Entry cards prompt users for a string of text or numbers. They allow you to determine the format of the string the user enters and to place default text in the entry field.

The syntax for entry card (<ENTRY>) statements is:

```
<ENTRY name=card_name format=fmt default=def key=arg
  ENTRYTYPE=entry_mode>
  actions
  display_text
</ENTRY>
```

Element	Meaning
<i>name=card_name</i>	Specifies a name that allows you to navigate to the card from other cards.
<i>format=fmt</i>	Format specifiers for the data the user enters. If you omit this option, the phone allows the user to enter any type of data. For a complete list of format specifiers, see the <i>HDML Language Reference</i> .
<i>default=def</i>	A default value that appears in the entry field. The user can edit this value.
<i>key=arg</i>	The argument name the phone uses to return the data entered by the user.
<i>ENTRYTYPE=entry_mode</i>	Specifies whether the phone accepts mixed case text (TEXT) or all uppercase text (DEFAULT).
<i>actions</i>	The actions to execute when the user presses function keys.
<i>display_text</i>	The prompt the card displays.

The *format* option allows you to specify whether individual characters the user enters must be alphabetic (A), numeric (N), or alphanumeric (X). For example, the format specifier *NAAX* specifies that the user must enter a digit, followed by two alphabetic characters, followed by an alphanumeric character. To allow multiple characters of a specified type, you specify an asterisk (\*) followed by the type. For example, *NN\*A* specifies that the user can enter two digits followed by any number of alphabetic characters.

The *format* option also allows you to place automatic characters, which the user cannot edit, into the entry field. To do this, you add the characters between the other format specifier characters, preceding each character with a backslash (\). When the user enters data, the phone automatically inserts the characters at the specified positions. This is useful for ensuring that data such as phone numbers include parentheses or dashes in the correct places.

Suppose you specify the format specifier `\(NN\) -`. This format specifier instructs the phone to automatically insert a left parenthesis before the user has entered anything and a right parenthesis and a dash after the user enters two digits. The following table summarizes this behavior.

User action	Text that appears in entry field
No action	(
User enters a 1	(1
User enters a 2	(12) -
User backs up (erases) one digit	(1

For example, the following HDML deck generates the display shown in Figure 2-2:

```
<HDML VERSION=0.1.0 DECKNAV=NOAUTOLOAD>
  <ACTION TYPE=ACCEPT GOARGS="?">
    <ENTRY NAME=ssentry KEY=SSNUM FORMAT="\N\O\:\ NN\ -NN\ -NNNN"
      ENTRYTYPE=DEFAULT>
      "Social Security #:"
    </ENTRY>
</HDML>
```

Social Security #:  
 NO: 519-  
 OK

FIGURE 2-2. Partially completed entry card with automatic characters

Note that the `<ACTION>` statement in the deck listed above uses the `GOARGS` option instead of the `GO` option used in previous examples. The `GOARGS` option instructs the phone to load a specified URL and append the arguments defined in the current deck to it. The question mark (?) instructs the phone to use the URL of the current service.

The `SSNUM` argument from the entry card is the only argument defined in this deck. Its value is provided by the data the user enters in the card. For example, if the user enters `519-12-3456` and the service's number is `1000`, the phone requests the following URL:

```
:1000?SSNUM=519-12-3456
```



## Using choice cards

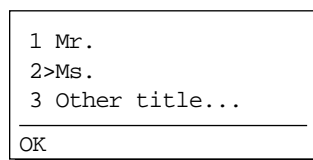
Choice cards prompt the user to choose an item from a list of items. The syntax for choice card `<CHOICE>` statements is:

```
<CHOICE name=card_name method=choice_method key=arg default=def>
  display_text
  actions
  <CE VALUE=val tasks>text
  <ce value=val tasks>text
  . . .
</CHOICE>
```

Element	Meaning
<i>name=card_name</i>	Specifies a name that allows you to navigate to the card from other cards.
<i>method=choice_method</i>	Specifies the list type. To make the phone automatically number the list, specify <code>LIST</code> . To leave the list unnumbered, specify <code>ALPHA</code> . If you do not specify a value for the <i>method</i> option, the phone uses the default ( <code>LIST</code> ).
<i>key=arg</i>	The argument name the phone uses to return the user's choice. The phone pairs this name with the value specified for the item the user chose. It adds the argument name-value pair to the argument list. See the following section for more information on argument lists.
<i>default=def</i>	The number of the default item. For no default item, specify <code>0</code> .
<i>display_text</i>	The prompt the card displays.
<i>actions</i>	The actions to execute when the user presses a function key.
<i>&lt;CE VALUE=val tasks&gt;text</i>	An individual choice item. <i>val</i> specifies the argument value the phone adds to the argument list when the user chooses the item. <i>tasks</i> specify tasks to execute when the user chooses the item. <i>text</i> specifies text of the item.

For example, the following HDML deck generates the display shown in Figure 2-3:

```
<HDML VERSION=0.1.0 DECKNAV=NOAUTOLOAD>
  <ACTION TYPE=ACCEPT GOARGS="?">
  <CHOICE KEY=TITLE DEFAULT=2>
    <CE VALUE=Mister>Mr.
    <CE VALUE=Ms>Ms.
    <CE GO="#titleentry">Other title...
  </CHOICE>
  . . .
</HDML>
```



1 Mr.  
2>Ms.  
3 Other title...

OK

FIGURE 2-3. Choice card

Note that the GO action defined for the third choice in this example overrides the `<ACTION>` statement for the deck. If the user chooses one of the first two items in the choice list, the phone requests the service URL with the argument `?TITLE=Mister` or `?TITLE=Ms`, respectively. If the user chooses the third item, the phone displays the card identified by the name `titleentry`.

## Building an argument list with multiple cards

Suppose you want to allow users to enter a set of associated data, much as they might enter data on an HTML form. You could do this by sequentially displaying decks containing entry or choice cards and saving the results returned for each deck. However, it is often simpler and more efficient to include all the choice and entry cards in a single deck.

As the user traverses the choice and entry cards in a deck, the phone appends the argument key-value pair from each card to a list. For example, the following deck contains two cards that create an argument list:

```
<HDML VERSION=0.1 DECKNAV=NOAUTOLOAD>
  <CHOICE NAME=titlechoice KEY=TITLE DEFAULT=2>
    <ACTION TYPE=ACCEPT GO="#nameentry">
    <CE VALUE=Mister>Mr.
    <CE VALUE=Miz>Ms.
  </CHOICE>
  <ENTRY NAME=nameentry KEY=USERNAME ENTRYTYPE=TEXT>
    <ACTION TYPE=ACCEPT GOARGS="?">
    Name:
  </ENTRY>
</HDML>
```

If the user chooses `Mr.` in the first card and enters `Marples` in the second card, the argument list looks as follows:

```
TITLE=Mister
USRNAME=Marples
```

When the user presses `ACCEPT` in the second card, the phone requests the service's URL with the argument list appended. For example, if the service number is 1000, the request would be:

```
:1000?TITLE=Mister&USRNAME=Marples
```

The argument list remains the same until the phone requests a new deck or a card resets an argument. When the phone requests a new deck, it clears the argument list.

### Resetting arguments in the argument list

HDML allows a card to reset an argument that has already been set by another card. If a card specifies an argument key that is already in the list, the phone uses the value from the card to replace the argument's current value. For example, the following deck includes two cards that set the same argument:

```
<HDML VERSION=0.1 DECKNAV=NOAUTOLOAD>
  <CHOICE NAME=titlechoice KEY=TITLE DEFAULT=2>
    <ACTION TYPE=ACCEPT GO="#titlentry">
      <CE VALUE=Mister>Mr.
      <CE VALUE=Miz>Ms.
    </CHOICE>
  . . .
  <ENTRY NAME=titleentry KEY=TITLE ENTRYTYPE=TEXT>
    <ACTION TYPE=ACCEPT GOARGS="?">
      Title:
    </ENTRY>
</HDML>
```

If the user chooses `Mr.` in the first card of this deck and types `HRH` in the second card, the resulting argument list is:

```
TITLE=HRH
```

### Providing easy navigation for data entry

When you create a deck with multiple interactive cards, it is important to provide a consistent navigation scheme that allows the user to back up through the deck without losing entered data. The default action for the CANCEL button is to return to the previous deck, which clears the argument list. This is usually not a desirable action—particularly if the user has entered several cards of data. To avoid this undesirable behavior, specify an `<ACTION>` statement for each card after the first card in the deck. Define the `<ACTION>` statement so that it instructs the phone to return to the previous card when the user presses CANCEL. For example, the following code creates the deck depicted in Figure 2-4.

```
<HDML VERSION=0.1.0 DECKNAV=NOAUTOLOAD>
  <CHOICE NAME=titlechoice KEY=TITLE DEFAULT=3>
    <ACTION TYPE=ACCEPT GO="#nameentry">
      <CE VALUE=Mister>Mr.
      <CE VALUE=Ms>Ms.
      <CE VALUE=HRH>Her Royal Highness
    </CHOICE>
    <ENTRY NAME=nameentry KEY=USERNAME ENTRYTYPE=TEXT>
      <ACTION TYPE=CANCEL GO="#titlechoice">
      <ACTION TYPE=ACCEPT GO="#occupentry">
      Name:
    </ENTRY>
    <ENTRY NAME=occupentry KEY=OCCUP ENTRYTYPE=TEXT>
      <ACTION TYPE=CANCEL GO="#nameentry">
      <ACTION TYPE=ACCEPT GOARGS="?">
      Occupation:
    </ENTRY>
  </HDML>
```

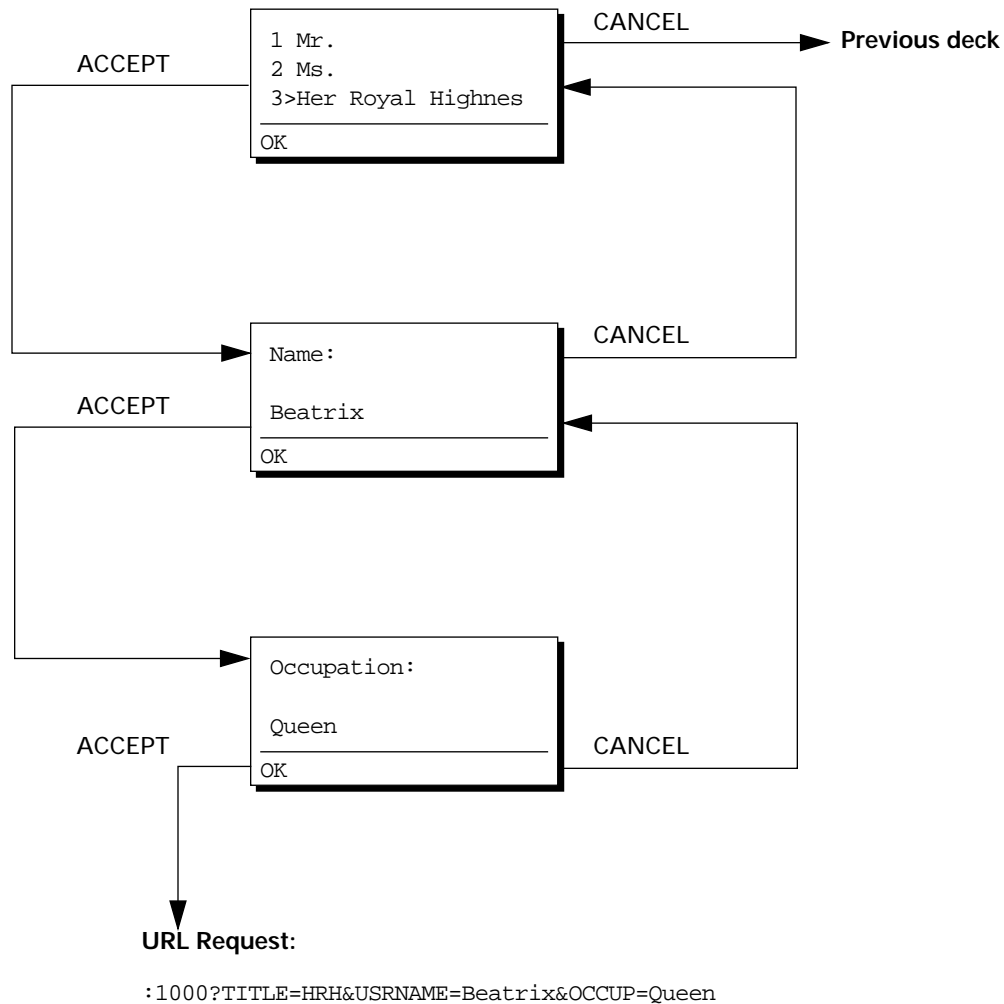


FIGURE 2-4. Deck with multiple interactive cards

If the user reaches the Occupation entry card and then presses CANCEL, the phone returns to the Name entry card. If the user presses CANCEL again, the phone returns to the Title choice card. This does not affect any of the arguments in the argument list. To change the list, the user must enter a different value or choose a different option in one of the cards and press ACCEPT.

## Modifying deck navigation

The following sections describe how to modify how the phone navigates among decks that your service generates.

### Working with deck history and deck caching

Like a conventional web browser, a UP.Phone maintains a history list and a cache. The history list is a list of the deck URLs the phone has displayed most recently. The phone maintains the history as a Last-In-First-Out (LIFO) list so that when users navigate backward they see the most recently visited decks first. The size of the history list is dependent on the phone model. When the history list reaches the phone's memory limitations, the phone drops URLs from the bottom of the history list.

The cache contains a list of recently requested URLs and the decks associated with them. It enables the phone to quickly redisplay a deck without requesting it again from the UP.Link. When the cache reaches the phone's memory limitations, the phone drops older decks from it.

When the user navigates backward, the phone traverses down the history list. Each time the user backs up a deck, the phone pops the deck it is currently displaying from the top of the history list and displays the deck that was formerly underneath it. The browser does not reload the deck from the UP.Link. Instead, it simply retrieves the deck from the cache.

### Example of deck caching

Suppose you provide a very simple stock quote service that is registered as service 1010. The home deck for the service is the following:

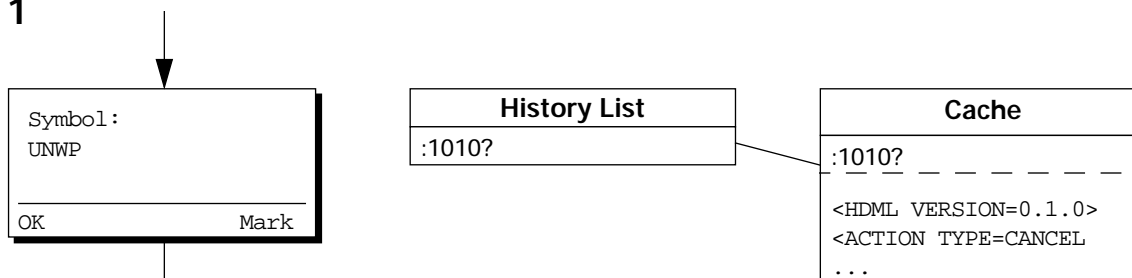
```
<HDML VERSION=0.1.0>
  <ACTION TYPE=CANCEL GOARGS="? ">
  <ENTRY KEY=TICKER ENTRYTYPE=TEXT>
    Symbol:
  </ENTRY>
</HDML>
```

After the user enters a stock ticker symbol and presses ACCEPT, the service displays a deck like the following:

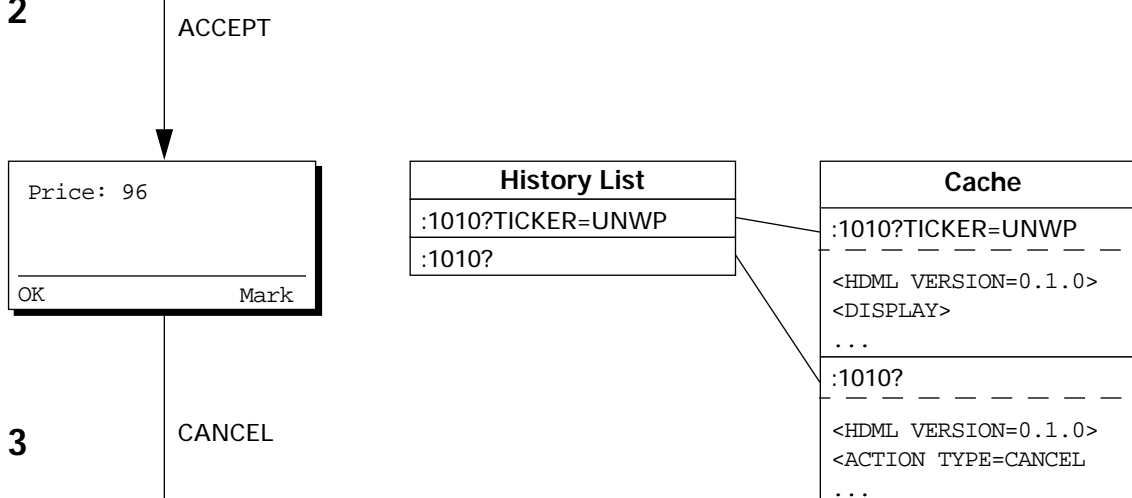
```
<HDML VERSION=0.1.0>
  <DISPLAY>
    Price: 96
  </DISPLAY>
</HDML>
```

Figure 2-5 summarizes the effects of user interactions with the service on the phone's history list and the cache. The graphics on the left depict the phone display after each interaction; the graphics on the right depict the corresponding states of the history list and the cache.

### Step 1



### Step 2



### Step 3

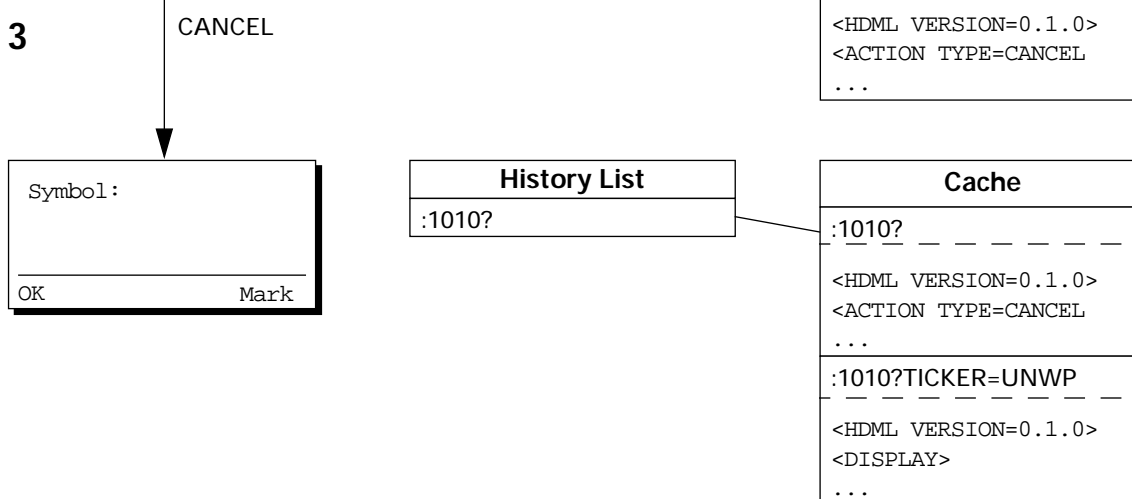


FIGURE 2-5. Navigating through the deck history

The following steps summarize how the user navigates the service and how it affects the history list and the cache.

- 1 The user navigates to the service's home deck (the deck that prompts the user to enter a stock ticker symbol).

The phone requests the service's home deck from the UP.Link and displays it. It adds the deck's URL to the history list; it adds the deck's HDML to the cache.

- 2 The user enters a ticker symbol (UNWP) and presses ACCEPT.

The phone requests the deck containing the stock price from the UP.Link and displays it. The phone adds the deck's URL to the history list and the deck's HDML to the cache.

- 3 The user presses CANCEL.

The phone returns to the previous deck in the history list (the deck that prompts the user for a stock symbol). It retrieves the deck's HDML from the cache and displays it.

Note that when the user navigates backward to a previous deck, the phone retrieves the previous deck from the cache. As a general rule, when the user backs up to a deck by pressing CANCEL, the UP.Phone always retrieves the deck from the cache.<sup>1</sup>

Also note that as the user traverses back from each deck, the phone removes the deck from the history list. This process is known as *pruning*. The phone doesn't support the WWW browser concept of a "forward" key that allows the user to proceed forward in the history list. The current deck is always at the top of the history list. To return to a deck that they have just backed up from, users may have to recreate whatever input was initially required to display the deck. In the current example, once users have traversed back to the deck that requests the stock symbol, they must reenter the UNWP symbol and press ACCEPT to see the deck with the UNWP stock price again.

As the user navigates backward, the phone *does not* remove decks from the cache. If, after the interactions shown in Figure 2-5, the user enters the stock ticker symbol UNWP again in the deck that requests a stock symbol, the phone retrieves the stock price deck (specified by the URL :1000?TICKER=UNWP) from the cache.

Because stock quotes are timely information, it is normally not desirable for the phone to retrieve them from the cache. The following section describes how to instruct the phone to reload certain decks from the UP.Link instead of the cache.

---

1. The only time the phone does not retrieve the previous deck from the cache is when the HDML overrides the CANCEL action at the deck level. Overriding the CANCEL action for individual cards is permissible, and is, in fact, recommended for decks containing multiple entry or choice cards. However, Unwired Planet strongly discourages overriding the CANCEL action at the deck level, because it makes navigation unpredictable for the user.



### Setting the cache time for a deck

When the user navigates to a URL that is already in the cache, the phone determines whether to retrieve the deck from its cache or to request the deck again from the UP.Link. It does this by comparing the cache time specified in the deck's HDML to the time elapsed since it last requested the deck. If the elapsed time is greater than the specified cache time, the phone automatically requests the deck again. Otherwise, it just displays the cached deck.

You can set the cache time for a deck by setting the `TTL` option in its `<HDML>` statement. The `TTL` option specifies the time to cache the deck in seconds.

If a deck is particularly time sensitive, set the cache time to `0` so that the phone requests the deck every time the user traverses it. If you don't set the `TTL` option for a deck, the phone uses the default cache time, which is 30 days.

---

**IMPORTANT** In version 0.1.0 of HDML, the phone rounds the cache time down to the nearest day.

---

### Bookmarks

UP.Phones allow users to bookmark decks that they intend to reload frequently. A bookmark is a shortcut that allows the user to navigate directly to a specified URL. For example, a user could bookmark the stock price deck shown in Figure 2-5. The phone stores the deck's URL along with a bookmark name the user chooses. The user can subsequently go to the bookmark list and choose the bookmark. When the user chooses the bookmark, the phone loads the URL.

In some cases you may want to prevent the user from bookmarking a deck. To do this, you set the deck's `DECKNAV` option to `NOAUTOLOAD` (as most of the examples in this chapter do). If a deck has the `DECKNAV` option set to `NOAUTOLOAD`, the phone deactivates the Mark key.

---

**IMPORTANT** Always set the `DECKNAV` option to `NOAUTOLOAD` for decks that execute or confirm transactions that the user might not want to repeat.

---

For example, a deck confirming a securities trade might have a URL similar to the following:

```
:1277?ACTION=buy&TICKER=AMER&SHARES=5000&PRICE=73.5
```

If you do not set the `DECKNAV` option to `NOAUTOLOAD`, the user could bookmark the deck and return to it. If the deck's cache time has been exceeded, it could result in the service executing the trade again—which could be disastrous.

---

## Replacing a deck in the cache

Each time the phone requests and receives a different deck URL, it adds the URL and the deck to the cache. If a deck has the same service ID and path as a deck already in the cache, but has slightly different parameters, the phone considers it a different deck and adds it to the cache. Sometimes this behavior is desirable because each deck is actually different. However, in some cases, the cache may accumulate multiple instances of what is really the same deck.

You can prevent the cache from accumulating multiple instances of a deck by setting the `DECKID` option in the deck's `<HDML>` statement. The `DECKID` option must specify a URL that is relative to the service's URL. When the phone loads the deck, it checks the cache for a deck with the same ID. If it finds one, it replaces it with the new deck.

---

## Formatting display text

The following sections describe how to use HDML statements and tags to format text that is displayed on the phone.

---

### Adding line breaks

To start a new line in display text, insert a `<BR>` statement where you want to start the line. Note that simply adding a carriage return or line feed in the HDML has no effect; the HDML compiler treats it as a single space.

---

### Setting text wrapping and horizontal scrolling

There are two ways UP.Phones can handle text lines that exceed the display width:

- Wrapping—the phone spills extra text onto subsequent lines in the display
- Horizontal or “Times Square” scrolling—the phone displays the entire text line on one display line and periodically scrolls the line from left to right so the user can see the entire line

Wrapping is the default mode for all display text except the text in individual choice entry elements. To change the mode to wrapping, insert a `<WRAP>` statement in the text at the point where you want to change it. To change the mode to horizontal scrolling, insert a `<LINE>` statement. Once you set the mode, it affects subsequent lines until you reset it. Each `<WRAP>` and `<LINE>` statement starts a new line.

When the phone wraps a line of text, it attempts to wrap between words. To prevent the phone from wrapping between two words, separate the words with an `&nbsp;` (nonbreaking space) escape sequence instead of a space.

For example, the following deck displays the card in Figure 2-6:

```
<HDML VERSION=0.1.0 DECKNAV=NOAUTOLOAD>
  <DISPLAY>
    <LINE>This line is scrolled horizontally.
    <BR>This line is also scrolled horizontally.
    <WRAP>This line is wrapped&nbsp;early&nbsp;due to
      nonbreaking spaces.
  </DISPLAY>
</HDML>
```

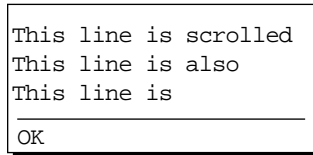


FIGURE 2-6. Display with horizontally scrolled and wrapped lines

## Specifying text alignment and tabs

By default, formatted text is left aligned. To right align a line, precede it with a `<RIGHT>` statement. To center it, precede it with a `<CENTER>` statement. `<RIGHT>` and `<CENTER>` statements apply only to the current line. Center and right alignment are not allowed in items on choice cards.

To create aligned columns of text, insert `<TAB>` statements in the text. The phone automatically sets tab stops for you. It treats contiguous lines containing tabs as rows in a table, setting the number of columns to the number of tabs in the line with the most tabs. It sets the width of each column to accommodate the column's largest cell. Using tabs in centered lines has unpredictable results; text columns may run into each other.

For example, the following code defines the display card shown in Figure 2-7:

```
<HDML VERSION=0.1.0 DECKNAV=NOAUTOLOAD>
  <DISPLAY>
    <CENTER>Highway speed
    <LINE><TAB>I80<TAB>US50
    <LINE>2pm<TAB>62mph<TAB>51mph
    <LINE>3pm<TAB>58mph<TAB>50mph
  </DISPLAY>
</HDML>
```

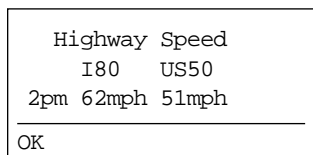


FIGURE 2-7. Display card with formatted text containing tabs

## Displaying special characters

HDML reserves the `<`, `>`, `"`, and `&` characters. To display one of these characters in formatted text, you must specify one of the following escape sequences.

Character	Escape sequence
<code>&lt;</code>	<code>&amp;lt;</code>
<code>&gt;</code>	<code>&amp;gt;</code>
<code>"</code>	<code>&amp;quot;</code>
<code>&amp;</code>	<code>&amp;amp;</code>

**IMPORTANT** The semicolon (`;`) is part of the escape sequence for a special character. If you omit it, the HDML compiler generates an error message.

For example, the following code displays the card in Figure 2-8:

```
<HDML VERSION=0.1.0 DECKNAV=NOAUTOLOAD>
  <DISPLAY>
    <LINE>&quot;&lt;BR&gt;S&amp;P&quot;
    <BR>displays:
    <BR>S&amp;P
  </DISPLAY>
</HDML>
```

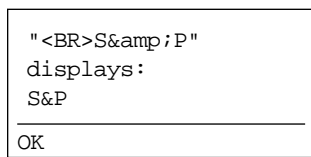


FIGURE 2-8. Display card containing special characters

The UP.Link Developer's Kit provides a Perl utility function named `HDMLEscapeString()`, which converts the HDML reserved characters in a string to valid escape sequences. For example, the following code:

```
. . .
require 'HDMLOutput.pl';
$estr = &AppUtils'HDMLEscapeString("<BR>");
print $estr;
```

prints:

```
&lt;BR&gt;
```

For more information on this function, see the *HDML Language Reference*.

## Choosing the optimal deck size

Because there is a certain amount of overhead involved in processing each request from an UP.Phone, you can often improve performance by increasing the amount of information your service transmits in each deck. Because HDML is designed to be compact, adding an additional card or two to a deck usually does not noticeably increase the time it takes to transmit the deck to the phone.

For example, suppose your service lists several movie titles and allows the user to retrieve a brief description of a movie by choosing its title. You could implement the service by generating one deck that lists the movie titles and a separate deck for each movie description, as follows:

```
<HDML VERSION=0.1.0 DECKNAV=NOAUTOLOAD>
  <ACTION TYPE=ACCEPT GOARGS="?">
    <CHOICE NAME=titlechoice KEY=TITLE>
      <CE VALUE=Harold>Harold and Maude
      <CE VALUE=Balloon>The Red Balloon
    </CHOICE>
  </HDML>
. . .
<HDML VERSION=0.1.0>
  <DISPLAY>
    Harold and Maude is an offbeat 60s-style comedy.
  </DISPLAY>
</HDML>
. . .
<HDML VERSION=0.1.0>
  <DISPLAY>
    The Red Balloon is a whimsical French movie.
  </DISPLAY>
</HDML>
. . .
```

However, this approach requires the user to generate at least two UP.Phone requests: one to get the list of movies and one to get the description of each movie. It is more efficient to put the movie list and the movie descriptions in a single deck as follows:

```
<HDML VERSION=0.1.0 DECKNAV=NOAUTOLOAD>
  <CHOICE NAME=titlechoice KEY=TITLE>
    <CE GO="#Harold">Harold and Maude
    <CE GO="#Ballon">The Red Balloon
  </CHOICE>
  <DISPLAY name=Harold>
    <ACTION TYPE=CANCEL GO="#titlechoice">
    <ACTION TYPE=ACCEPT GO="#titlechoice">
    Harold and Maude is an offbeat 60s-style comedy.
  </DISPLAY>
  <DISPLAY name=Balloon>
    <ACTION TYPE=CANCEL GO="#titlechoice">
    <ACTION TYPE=ACCEPT GO="#titlechoice">
    The Red Balloon is a whimsical French movie.
  </DISPLAY>
</HDML>
. . .
```

When you use this approach, the user needs to generate only one phone request. After the phone receives the deck, the user can look at the description of either movie without generating an additional phone request. When the user chooses a movie in the choice card, there is no perceptible delay before the phone displays the description, because the description is already in the phone.

There is, however, a limitation to how large you can make a deck.

---

**IMPORTANT** An UP.Link cannot transmit *compiled* HDML decks larger than 1492 bytes to an UP.Phone. If you send a compiled deck larger than 1492 bytes to the UP.Link, the UP.Link generates an error.

---

Compiling a deck reduces its size; an *uncompiled* HDML deck smaller than 1500 bytes will always yield a *compiled* deck that is smaller than 1492 bytes. The percentage size reduction achieved by compiling a deck varies from deck to deck—it depends on the HDML tags used in the deck. There is no way to determine the exact compiled size of a deck except by compiling it. Always check the size of a compiled deck if its uncompiled HDML is larger than 1500 bytes.



## Creating UP.Link Services

---

# 3

This chapter discusses issues involved in creating an UP.Link service. It provides hints on writing the application that implements your service and instructions on compiling HDML and testing your service.

### Writing an application to implement a service

---

An UP.Link service can be implemented by any web server application that generates valid HDML in response to requests from the UP.Link. Most of the examples in this manual are for applications written in Perl. However, your application need not be written in Perl.

#### Using HTTP headers

---

It is very important to use the correct HTTP header when you send an HTTP response to the UP.Link. If your application uses Netscape CGI conventions, it should prepend the following header to each compiled HDML deck:

```
Content-type: application/x-hdmlc
```

```
. . . . .
```

Note that the header must be followed by a blank line.

If your application does not use Netscape CGI conventions, it should use the following header:

```
HTTP/1.0 200
Content-type: application/x-hdmlc
```

```
. . . . .
```

This header must also be followed by a blank line. Applications that do not use Netscape CGI conventions but are running under a Netscape server should be named according to the convention, `nph-appname.cgi`.



If you are using Perl to create your service, you can use UP.Link Developer's Kit Perl utility functions to automatically prepend HTTP headers to your HTTP responses. For more information, see "Using Perl utilities to compile HDML" on page 41.

## Using URL arguments added by the UP.Link

---

When the UP.Link relays an HTTP request from a phone to your service, it adds some arguments to the URL. These arguments provide the following information about the request.

Argument	Meaning
UPSubNo	The subscriber number of the UP.Link phone subscriber originating the request.
UPSvcNo	The number of the requested service.
UPDevice	The phone or handheld device type. For a list of supported devices, see the Unwired Planet web pages at: <a href="http://www.upplanet.com">http://www.upplanet.com</a>
UPHdmlcVersion	The HDML version number. The format of this number is <i>maj.min</i> , where <i>maj</i> is the major version number and <i>min</i> is the minor version number; for example, 1.0.

Your application can use these arguments to tailor its response to a request. For example, if your application is registered to provide several services, it can use the `UPSvcNo` argument to identify which service the user has requested.

The UP.Link Developer's Kit provides a Perl function named `ParseCGIVars()` that makes it easy to retrieve these arguments. For example, the following code prints the UP.Link subscriber number:

```

. . .
require 'CGISupport.pl';
. . .
%cgiVars = &AppUtils'ParseCGIVars;
print $cgiVars{"UPSubNo"};

```

For information on this and other Perl utility functions, see the *HDML Language Reference*.

## Generating compiled HDML

The following sections describe different ways you can compile your application's HDML output into compiled HDML, a binary-encoded format that the UP.Link understands.

---

**IMPORTANT** Each HTTP response from your application to the UP.Link must contain *compiled* HDML. If an HTTP response contains uncompiled HDML, the UP.Link displays an “Invalid Service Response” error message on the phone that originated the request.

---

### Using Perl utilities to compile HDML

To create an HTTP response to a request from the UP.Link, you can pipe output from your service to the HDML compiler and then append the HTTP header to it. However, if you are using Perl to create your service, it is normally easier to call the `OutputDeck()` function provided in the `HDMLOutput.pl` file. This function compiles an HDML deck and automatically adds the HTTP header to it.

For example, the following code:

```
#!/usr/local/bin/perl
#
require 'HDMLOutput.pl';

# Uncomment the following line if you're not using Netscape
# CGI conventions
# $AppUtils'CGI = $HDMLOutput'CGIStandard;
$DECK =
    "<HDML VERSION=0.1.0 TTL=0 DECKNAV=NOAUTOLOAD>".
    "<DISPLAY>".
    "<LINE>The Wuggly Ump is far away".
    "</DISPLAY>".
    "</HDML>";
&AppUtils'OutputDeck($DECK);
```

creates an HTTP response containing binary data that looks similar to the following:

```
Content-type: application/x-hdmlc

Ñ 0±The©Wuggly©Ump©is©far©away
```

For reference information on the `OutputDeck()` function, see the *HDML Language Reference*.

## Using the C++ HDML compiler library

The UP.Link Developer's Kit provides C++ classes that allow you to compile HDML. The following sections describe how to use these classes. For complete reference information on the classes, see the *HDML Language Reference*.

The Developer's Kit includes the following individual classes.

Class	Purpose
<code>TpcParser</code>	Compiles textual HDML into an Abstract Syntax Tree (AST).
<code>TpcParseError</code>	Encapsulates errors generated by the <code>TpcParser</code> class.
<code>TpcHDMLCGen</code>	Converts an AST into compiled HDML.
<code>TpcHDMLGen</code>	Converts an AST into HDML (uncompiled).

### Using the classes

To compile HDML with the HDML compiler C++ classes, you follow these general steps:

- 1 Parse the uncompiled HDML into an Abstract Syntax Tree (AST) object.
- 2 If there are no errors, convert the AST object into compiled HDML.
- 3 If there are errors, log the errors.

Once the HDML is compiled into an AST, you can convert it to compiled HDML or back to uncompiled HDML. Normally, you will need to convert an AST back to uncompiled HDML only for debugging.

The HDML compiler takes care of its own memory allocation and deallocation: any objects it creates internally it destroys internally. If a class that calls the HDML compiler classes creates an object, it should also destroy the object.

### Compiling and linking your application with the HDML compiler classes

To use the HDML compiler classes in your application, include the `hdmlc.h` header file provided with the UP.Link Developer's Kit. On UNIX, link your application with the `hdmlc.so` library. On Windows NT, link it with `hdmlc.lib` library.

### Example

The following code reads HDML from standard input and generates compiled HDML to standard output. Following the code is a line-by-line explanation of how it works.

```

1  #include <iostream.h>
2  #include "hdmlc.h"
3
4  int main()
5  {
6      TpcParser parser("Standard Input", cin);
7      if (parser.Parse())
8      {
9          TpcHDMLCGen gen(cout);
10         parser.Generate(&gen);
11     }
12     else
13     {
14         TpcParseError *err;
15         err = parser.GetErrors();
16         while (err)
17         {
18             cerr << err << endl;
19             err = err->GetNext();
20         }
21     }
22 }

```

Line 2 includes the header file containing the HDML compiler classes.

Line 6 creates a parser object. The first argument specifies the name of the file the HDML is read from. This filename is used only to generate error messages: each error message is preceded by the filename. The second argument instructs the `TpcParser` object to read from the `cin` iostream (standard input).

Line 7 instructs the parser object to parse the input. If it succeeds, it returns `TRUE`; otherwise, it returns `FALSE`.

Lines 8-11 output the HDML if the parser succeeds in parsing the HDML input into an AST. Line 8 creates a `TpcHDMLCGen` generator object and sets its output to the standard output iostream (`cout`). Line 10 instructs the parser to generate output. The `TpcParser::Generate()` method scans the AST, asking the generator object to output the AST in compiled HDML.

Lines 13-20 output error messages, if the parser fails to parse the HDML into an AST. Line 14 declares a pointer to an error object. The error class is a linked list of errors. Line 15 uses `TpcParser::GetErrors()` method to get the head of the error list. Lines 17-20 traverse the list of errors, printing each error. Each `TpcParserError` object provides an interface to write itself to the iostream.

---

## Using the command-line HDML compiler

---

The UP.Link Developer's Kit provides command-line programs that compile and decompile HDML; refer to the following table. These programs are useful if you want to generate compiled HDML from applications that are not written in C++ or Perl, or if you want to test static HDML decks.

Program	Description
hdmlc	HDML compiler for UNIX
hdmlc.exe	HDML compiler for Windows NT
hdml d	HDML decompiler for UNIX
hdml d.exe	HDML decompiler for Windows NT

The HDML compiler and decompiler support the standard input conventions of UNIX and Windows NT. For example, on UNIX, either of the following commands will compile the HDML in the file `my.hdml` into the file `my.hdmlc`:

```
cat my.hdml | hdmlc > my.hdmlc
hdmlc my.hdml > my.hdmlc
```

When they encounter invalid input, the HDML compiler and decompiler generate error messages specifying the filename, the line at which each error is encountered, and a general description of the error. For a list of the error messages, see the *HDML Language Reference*.

---

## Retrieving web pages

---

The UP.Link Developer's Kit provides Perl utilities that make it easy to connect to HTTP sites and retrieve web pages. The following table lists these utilities and their use.

Utility function name	Use
<code>HTTPConnect()</code>	Connects a specified URL to a socket.
<code>HTTPConnectAndBuffer()</code>	Retrieves the content of a specified URL and stores it to a buffer.
<code>HTTPConnectAndMatch()</code>	Returns the first occurrence of a regular expression in the content of a specified URL.
<code>HTTPDisconnect()</code>	Disconnects the current URL from a specified socket.
<code>HTTPSendRequest()</code>	Requests a web page from a specified socket.
<code>URLConnect()</code>	Connects a socket to the specified port or service of a specified host.
<code>URLParse()</code>	Parses a URL into a service protocol, a hostname, and a filepath.

For example code and information on these utilities, see Chapter 2, "HDML Perl Utilities Reference" in the *HDML Language Reference*.

